

## Magic IT Solutions

---

- Magic IT Solutions
- LinkedIn
- Facebook
- Currículo
- Login


## Recherche

---

OK

Like

Be the first of your friends to like this.



Add a comment...

☒ Post to Facebook

Posting as JI Vaz (Not you?)

Comment

Facebook social plugin

## Expressões regulares

---

Em ciência da computação, uma expressão regular (ou o estrangeirismo regex, abreviação do inglês regular expression) provê uma forma concisa e flexível de identificar cadeias de caracteres de interesse, como caracteres particulares, palavras ou padrões de caracteres. Expressões regulares são escritas numa linguagem formal que pode ser interpretada por um processador de expressão regular, um programa que ou serve um gerador de analisador sintático ou examina o texto e identifica partes que casam com a especificação dada.

O termo deriva do trabalho do matemático norte-americano Stephen Cole Kleene, que desenvolveu as expressões regulares como uma notação ao que ele chamava de álgebra de conjuntos regulares. Seu trabalho serviu de base para os primeiros algoritmos computacionais de busca, e depois para algumas das mais antigas ferramentas de tratamento de texto da plataforma Unix.

O uso atual de expressões regulares inclui procura e substituição de texto em editores de texto e linguagens de programação, validação de formatos de texto (validação de protocolos ou formatos digitais), realce de sintaxe e filtragem de informação.

## Conceitos básicos

Uma expressão regular (ou, um padrão) descreve um conjunto de cadeias de caracteres, de forma concisa, sem precisar listar todos os elementos do conjunto. Por exemplo, um conjunto contendo as cadeias "Handel", "Händel" e "Haendel" pode ser descrito pelo padrão `H(ä|ae?)ndel`. A maioria dos formalismos provê pelo menos três operações para construir expressões regulares.

A primeira delas é a alternância, em que uma barra vertical (`|`) separa alternativas. Por exemplo, `psicadélico|psicodélico` pode casar "psicadélico" ou "psicodélico". A segunda operação é o agrupamento, em que parênteses (`()`) são usados para definir o escopo e a precedência de operadores, entre outros usos. Por exemplo, `psicadélico|psicodélico` e `psic(a|o)délico` são equivalentes e ambas descrevem "psicadélico" e "psicodélico". Por fim, a terceira operação é a quantificação (ou repetição). Um quantificador após um token (como um caractere) ou um agrupamento especifica a quantidade de vezes que o elemento precedente pode ocorrer. Os quantificadores mais comuns são `?`, `*` e `+`. O ponto de interrogação indica que há zero ou uma ocorrência do elemento precedente. Por exemplo, `ac?ção` casa tanto "acção" quanto "ação". Já o asterisco indica que há zero ou mais ocorrências do elemento precedente. Por exemplo, `ab*c` casa "ac", "abc", "abbc", "abbbc", e assim por diante. Por fim, o sinal de adição indica que há uma ou mais ocorrências do elemento precedente. Por exemplo, `ab+c` casa "abc", "abbc", "abbbc", e assim por diante, mas não "ac".

Essas construções podem ser combinadas arbitrariamente para formar expressões complexas, assim como expressões aritméticas com números e operações de adição, subtração, multiplicação e divisão. De forma geral, há diversas expressões regulares para descrever um mesmo conjunto de cadeias de caracteres. A sintaxe exata da expressão regular e os operadores disponíveis variam entre as implementações.

## História

A origem das expressões regulares estão na teoria dos autômatos e na teoria das linguagens formais, e ambas fazem parte da teoria da computação. Esses campos estudam modelos de computação (autômatas) e formas de descrição e classificação de linguagens formais. Na década de 1950, o matemático Stephen Cole Kleene descreveu tais modelos usando sua notação matemática chamada de "conjuntos regulares", formando a álgebra de Kleene. A linguagem SNOBOL foi uma implementação pioneira de casamento de padrões, mas não era idêntica às expressões regulares. Ken Thompson construiu a notação de Kleene no editor de texto QED como uma forma de casamento de padrões em arquivos de texto. Posteriormente, ele adicionou essa funcionalidade no editor de texto Unix `ed`, que resultou no uso de expressões regulares na popular ferramenta de busca `grep`. Desde então, diversas variações da adaptação original de Thompson foram usadas em Unix e derivados, incluindo `expr`, `AWK`, `Emacs`, `vi` e `lex`.

As expressões regulares de Perl e Tcl foram derivadas da biblioteca escrita por Henry Spencer, e no Perl a funcionalidade foi expandida posteriormente. Philip Hazel desenvolveu a PCRE (Perl Compatible Regular Expressions), uma biblioteca usada por diversas ferramentas modernas como PHP e o servidor Apache. Parte do desenvolvimento do Perl 6 foi melhorar a integração das expressões regulares de Perl, e aumentar seu escopo e funcionalidade para permitir a definição de gramáticas de expressão de analisadores sintáticos. O resultado foi uma mini-linguagem, as regras do Perl 6, usada para definir a gramática do Perl 6 assim como fornecer uma ferramenta para programadores da linguagem. Tais regras mantiveram as funcionalidades de expressões regulares do Perl 5.x, mas também permitiram uma definição BNF de um analisador sintático descendente recursivo.

O uso de expressões regulares em normas de informação estruturada para a modelagem de documentos e bancos de dados começou na década de 1960, e expandiu na década de 1980 quando normas como a ISO SGML foram consolidadas.

## Vou mostrar alguns metacaracteres que vamos usar ao decorrer do artigo

#Metacaractere	Nome
.	Ponto
[]	Lista
[^]	Lista negada

?	Opcional	
*	Asterisco	
+	Mais	
{ }	Chaves	
^	Circunflexo	
\$	Cifrão	
\b	Borda	
\	Escape	
	Ou	
()	Grupo	
\1	Retrovisor	
 #Metacaracteres Representates		
#Metacaracter	Nome	Função
.	Ponto	Um caracter qualquer
[...]	Lista	Lista de caracteres permitidos
[^...]	Lista negada	Lista de caracteres proibidos
 #Metacaracteres Quantificadores		
#Metacaracter	Nome	Função
?	Opcional	Zero ou um
*	Asterisco	Zero, um ou mais
+	Mais	Um ou mais
{n,m}	Chaves	de n até m
 #Metacaracteres Âncoras		
#Metacaracter	Nome	Função
^	Circunflexo	Início da linha
\$	Cifrão	Fim da linha
\b	Borda	Início ou fim de palavra
 #Metacaracteres Outros		
#Metacaracter	Nome	Função
\c	Escape	Torna literal o caractere c
	Ou	Ou um ou outro
(...)	Grupo	Delimita um grupo
\1...\9	Retrovisor	Texto casado nos grupos 1...9

## Metacaracteres tipo Representante

O primeiro grupo de metacaracteres que veremos são os do tipo representante, ou seja, são metacaracteres cuja função é representar um ou mais caracteres.

**Ponto:** O necessitado.

O ponto é nosso curinga solitário, que está sempre à procura de um casamento, não importa com quem seja. Pode ser um número, uma letra, um TAB, um @, o que vier ele traça, pois o ponto casa qualquer coisa.

```
#Tipos de uso.
Expressão      Casa com
n.o            não, nao, ...
.eclado        teclado, Teclado, ...
e.tendido      estendido, extendido, eztenddo, ...
12.45          12:45, 12 45, 12.45, ...
<.>            <B>, <i>, <p>, ...
```

Como testar vamos usar o comando echo que jogar a saída na tela então se eu usar echo "Douglas" vai aparecer somente Douglas na tela

Testando a primeira expressão com o casamento da expressão n.o

```
echo "não" | egrep "n.o"
não
```

Testando novamente o casamento da expressão n.o

```
echo "nao" | egrep "n.o"
nao
```

## Resumão

- O ponto casa com qualquer coisa
- O ponto casa com o ponto.
- O ponto é um curinga para se casar um caracter

**Lista:** A exigente[...]

Bem mais exigente que o ponto, a lista não casa com qualquer um. Ela sabe exatamente o que quer, e nada diferente daquilo, a lista casa com quem ela conhece.

Ela guarda dentro de si os caracteres permitidos para casar, então algo como [aeiou] limita nosso casamento a letras vogais.

```
#Tipo de uso.
Expressão      Casa com
n[ãa]o         não, nao
[Tt]eclado     Teclado, teclado
e[sx]tendido   estendido, extendido
12[:. ]45       12:45, 12.45, 12 45
<[BIP]>         <B>, <I>, <P>
<[BIPbip]>      <B>, <I>, <P>, <b>, <i>, <p>
```

**OBS:.** Dentro da lista, todo mundo é normal, então aquele ponto é simplesmente um ponto normal e não um metacaractere.

Temos uma execução a regra que todo mundo é normal, fora o traço. Se tivermos um traço (-) entre dois caracteres, isso representa todo o intervalo entre eles.

**EX:** [0123456789] é igual a [0-9]

O traço indica um intervalo, então 0-9 se lê: de zero a nove

Como o traço é especial dentro da lista, como fazer quando você quiser colocar na lista um traço literal?

Basta colocar o traço no final da lista, assim [0-9-] casa números ou um traço. E tem os colchetes, que são os delimitadores da lista como incluí-los dentro dela?

O colchete que abre não tem problema, pode colocá-lo em qualquer lugar na lista, pois ela já está aberta mesmo e não se pode ter uma lista dentro da outra.

O colchete que fecha deve ser colocado no começo da lista, ser o primeiro item dela, para não confundir com o colchete que termina a lista. Então []-] cada um ] ou um -.

Vamos juntar tudo e fazer uma lista que case ambos os colchetes e o traço: [][-].

Vamos testar a primeira expressão n[ãa]o

```
echo "nao" | egrep "n[ãa]o"
nao
```

Vamos a mais um teste sobre a primeira expressão n[ãa]o

```
echo "não" | egrep "n[ãa]o"
não
```

## Dominando caracteres acentuados (POSIX)

Já que diversos grupos de caracteres dependem de uma configuração de locale específica, a POSIX define algumas classes (ou categorias) de caracteres para fornecer um método padrão de acesso a alguns grupos específicos de caracteres bastante utilizados, como mostrado na seguinte tabela:

```
Classes de caracteres
[:alnum:]      Caracteres alfanuméricos, o que no caso de ASCII corresponde a [A-Za-z0-9].
[:alpha:]      Caracteres alfabéticos, o que no caso de ASCII corresponde a [A-Za-z].
[:blank:]      Espaço e tabulação, o que no caso de ASCII corresponde a [ \t].
[:cntrl:]      Caracteres de controle, o que no caso de ASCII corresponde a [\x00-\x1F\x7F].
[:digit:]      Dígitos, o que no caso de ASCII corresponde a [0-9].
[:graph:]      Caracteres visíveis, o que no caso de ASCII corresponde a [\x21-\x7E].
[:lower:]      Caracteres em caixa baixa, o que no caso de ASCII corresponde a [a-z].
[:print:]      Caracteres visíveis e espaços, o que no caso de ASCII corresponde a [\x20-\x7E].
[:punct:]      Caracteres de pontuação, o que no caso de ASCII corresponde a [-!"#$%&'()*+,-./:;<=>?@[\\]_`{|}~]."
```

```
[[:space:]]      Caracteres de espaços em branco, o que no caso de ASCII corresponde a [ \t\r\n\v\f].  
[[:upper:]]      Caracteres em caixa alta, o que no caso de ASCII corresponde a [A-Z].  
[[:xdigit:]]     Dígitos hexadecimais, o que no caso de ASCII corresponde a [A-Fa-f0-9].
```

Pode-se negar uma classe de caracteres precedendo um acento circunflexo ao nome da classe. Por exemplo, para negar `[[:digit:]]` usa-se `[[:^digit:]]`.

Para utilizar em uma ER temos que passar como no exemplo

```
egrep '[[[:lower:]]]' /etc/passwd
```

## Resumão

- A lista casa com quem ela conhece e tem suas próprias regras.
- Dentro da lista, todo mundo é normal
- Dentro da lista, traço indica intervalo.
- Um `-` literal deve ser o último item da lista.
- Um `]` literal deve ser o primeiro item da lista.
- Os intervalos repetam a tabela ASCII (não use A-z)
- `[[:classes POSIX:]]` incluem acentuação, A-Z não.

**Lista negada:** a experiente `[^...]`

Não tão exigente quando a lista nem tão necessitada quanto o ponto, temos a lista negada, que pelas suas más experiências passadas, sabe o que não serve para ela casar. A única diferença entre a lista e a lista negada é que ela possui lógica inversa, ou seja, ela casará com qualquer coisa, fora os componentes listados. Observe que a diferença em sua notação é o que o primeiro caractere da lista é um circunflexo, ele indica que esta é uma lista negada. Então se `[0-9]` são números, `[^0-9]` é qualquer coisa fora números.

## Resumão

- Uma lista negada segue todas as regras de uma lista normal.
- Um `^` literal não deve ser o primeiro item da lista.
- `[[:classes POSIX:]]` podem ser negadas.
- A lista negada sempre deve casar algo.

## Metacaracteres tipo Quantificador

Aqui chegamos ao segundo tipo de metacaracteres, os quantificadores que servem para indicar o número de repetições permitidas para a entidade imediatamente anterior. Essa entidade pode ser um caractere ou metacaractere.

Em outras palavras, eles dizem a quantidade de repetições que o átomo anterior que pode ter, quantas vezes ele pode aparecer.

Os quantificadores não são quantificáveis, então dois deles seguidos em uma ER é um erro, salvo quantificadores não-gulosos, que veremos depois.

E memorize, por enquanto sem entender o porquê: todos os quantificadores são gulosos.

### Opcional: o opcional ?

O opcional é um quantificador que não esquentar a cabeça, para ele pode ter ou não a ocorrência da entidade anterior, pois ele a repete 0 ou 1 vez. Por exemplo, a ER 7? significa zero ou uma ocorrência do número 7. Se tiver um 7, beleza, casamento efetuado. Se não tiver, beleza também. Isso torna o 7 opcional (daí o nome), que tendo ou não, a ER casa. Veja mais um exemplo, o plural. A ER ondas? tem a letra s marcada como opcional, então ela casa onda e ondas.

**Atenção:** Cada letra normal é um átomo da ER, então o opcional é aplicado somente ao s e não a palavra toda.

### Como ler uma ER

É bem simples, Primeiro lê-se átomo por átomo, depois entende-se o todo e depois se analisa as possibilidades. Na nossa ER fala[r!]? em questão, sua leitura fica: um f seguido de um a, seguido de um l, seguido de um a, seguido de: ou r, ou !, ambos opcionais.

Feita a leitura, agora temos de entender o todo, ou seja, temos um trecho literal fala, seguido de uma lista opcional de caracteres. Para descobrirmos as possibilidades, é o fala seguido de cada um dos itens da lista e por fim seguido por nenhum deles, pois a lista é opcional. Então fica:

# Expressão	Casa com
fala[r!]?	falar, fala!, fala

Pronto! Desvendamos os segredos da ER. É claro, esta é pequena e fácil, mas o que são ER grandes senão várias partes pequenas agrupadas ? O principal é dominar essa leitura por átomos. O resto é ler devagar até chegar ao final. Não há mistério.

### Resumão

- O opcional é opcional.
- O opcional é útil para procurar palavras no singular e plural.
- Podemos tornar opcionais caracteres e metacaracteres.
- Leia a ER átomo por átomo, da esquerda para a direita.
- Leia a ER, entenda o todo e analise as possibilidades.

### Asterisco: O tanto-faz

Se o opcional já não esquentar a cabeça, podendo ter ou não a entidade anterior, o asterisco é mais tranquilo ainda, pois para ele poder ter, não ter ou ter vários, infinitos. Em outras palavras, a entidade anterior pode aparecer em qualquer quantidade.

#Expressão	Casa com
7*0	0, 70, 770, 7770, ...
bi*p	bp, bip, biip, biiip, ...
b[ip]*	b, bi, bip, biipp, bpipipi, biiip, ...

### Testando a expressão

```
echo "biiiiiiiiiiiiiiip" | egrep "^bi*p$"
```

Como HTML é sempre um ótimo exemplo, voltamos ao nosso caso das marcações que podem ter vários espaços em branco após o identificador, então `<b >` e `</b >` são validos. Vamos colocar essa condição na ER:

#Expressão	Casa com
<code>&lt;/?[BIPbip] *</code>	<code>&lt;B&gt;, &lt;/B&gt;, &lt;/B &gt;, ..., &lt;p &gt;, ...</code>

Testando a expressão

```
echo "<b >" | egrep "</?[BIPbip] *>"
```

Note que agora com o asterisco, nossa ER já não tem mais um número finito de possibilidades. Vejamos como fica a leitura dessa ER: um `<`, seguido ou não de uma `/`, seguido de: ou `B`, ou `I`, ou `P`, ou `b`, ou `i`, ou `p`, seguido ou não de vários espaços, seguido de `>`.

## Apresentando a gulodice

Pergunta: o que casará `[ar]*a` na palavra arara? Alternativas:

- |          |   |
|----------|---|
| 1) a     | <code>[ar]</code> zero vezes, seguido de a.             |
| 2) ara   | <code>[ar]</code> duas vezes (a,r), seguido de a.       |
| 3) arara | <code>[ar]</code> quatro vezes (a,r,a,r), seguido de a. |
| 4) n.d.a |   |

Acertou se você escolheu a número 3. O asterisco repete em qualquer quantidade, mas ele sempre tentará repetir o máximo que conseguir. As três alternativas são válidas, mas entre casar a lista `[ar]` zero, duas ou quatro vezes, ele escolherá o maior número possível. Por isso se diz que o asterisco é guloso.

Essa gulodice às vezes é boa, às vezes é ruim. Os próximos quantificadores, mais e chaves, vem como o opcional já visto, são igualmente gulosos. Mais detalhes sobre o assunto, confira mais adiante.

## Apresentando o curinga .\*

Vimos até agora que temos dois metacaracteres extremamente abrangentes, como o ponto (qualquer caractere) e o asterisco (em qualquer quantidade). E se juntarmos os dois? Teremos qualquer caractere, em qualquer quantidade. Pare um instante para pensar nisso. O que isso significa, tudo ? Nada? A resposta é: ambos.

O nada, pois "qualquer quantidade" também é igual a "nenhuma quantidade". Então é opcional termos qualquer caractere, não importa. Assim, uma ER que seja simplesmente `.*` sempre será válida e casará mesmo uma linha vazia.

O tudo, pois "qualquer quantidade" também é igual a "tudo o que tiver". E é exatamente isso o que o asterisco faz, ele é guloso, ganancioso, e sempre tentará casar o máximo que conseguir.

**ATENÇÃO:** O curinga `.*` é qualquer coisa!



Assim, temos aqui o curinga das ERs, uma carta para se usar em qualquer situação. É muito comum ao escrever uma expressão regular, você definir alguns padrões que procura, e lá no meio, em uma parte que não importa, pode ser qualquer coisa, você coloca um `.*` e depois continua a expressão normalmente.

Por exemplo, para procurar ocorrência de duas palavras na mesma linha, relatório.\*amanhã serve para achar aquela linha maldita em que lhe pediram um trabalho "para ontem". Ou ainda, procurar acessos de um usuário em uma data qualquer: 12/06/2011.\*login.

## Resumão

- O asterisco repete em qualquer quantidade.
- Quantificadores são gulosos.
- O curinga `.*` é o tudo e o nada, qualquer coisa.

## Mais: O tem-que-ter

O mais tem funcionamento idêntico ao do asterisco, tudo o que vale para um, se aplica ao outro.

A única diferença é que o mais não é opcional, então a entidade anterior deve casar pelo menos uma vez, e pode ter várias.

Sua utilidade é quando queremos no mínimo uma repetição. Não há muito o que acrescentar, é um asterisco mais exigente...

#Expressão	Casa com
<code>7+0</code>	<code>70,770,7770,... 7777777777770,...</code>
<code>bi+p</code>	<code>bip, biip, biip, biip, biip,...</code>
<code>b[ip]+</code>	<code>bi,bip,biipp,bpipipi,biipiiip,bppp,...</code>

## Testando a expressão

```
echo "70" | egrep "7+0"
```

## Resumão

- O mais repete em qualquer quantidade, pelo menos uma vez.
- O mais é igual ao asterisco, só mais exigente.

## Chaves: O controle

Aqui Chaves não é o autor mexicano preferido de 10 entre 10 brasileiros. As chaves são a solução para uma quantificação mais controlada, onde se pode especificar exatamente quantas repetições se quer da entidade anterior.

Basicamente, `{n,m}` significa de `n` até `m` vezes, assim algo como `7{1,4}` casa `7,77,777` e `7777`. Só, nada mais que isso.

Temos também a sintaxe relaxada das chaves, em que podemos omitir a quantidade final, ou ainda, especificar exatamente um número:

#Metacaractere	Repetições
{1,3}	De 1 a 3
{3,}	Pelo menos 3 (3 ou mais)
{0,3}	Até 3
{3}	Exatamente 3
{1}	Exatamente 1
{0,1}	Zero ou 1 (igual ao opcional)
{0,}	Zero ou mais (igual ao asterisco)
{1,}	Um ou mais (igual ao mais)

## Testando a expressão

```
echo "7777" | egrep "^7{1,4}$"
```

Note que o {1} tem efeito nulo, pois 7{1} é igual a 7. Pode ser útil caso você queira impressionar alguém com sua ER, pode encher de {1} que não mudará sua lógica. Mas observe os três últimos exemplos.

Com as chaves, conseguimos simular o funcionamento de outros três metacaracteres, o opcional, o asterisco e o mais.

Se temos as chaves que já fazem o serviço, então para quê ter os outros três ? Você pode escolher a resposta que achar melhor Eu tenho algumas:

- \* é menor e mais fácil que {0,}
- As chaves foram criadas só depois dos outros
- \* Precisavam de mais metacaracteres para complicar o assunto.
- \*,+ e ? são links para as chaves.
- Alguns teclados antigos vinham sem a tecla {

Ah e sendo {0,} algo mais feio que um simples \*, isso também pode ser usado para tornar sua ER grande e intimidadora. Só cuidado para não atirar no próprio pé e depois não conseguir entender sua própria criação.

## Resumão

- Chaves são precisas
- Você pode especificar um número exato, um mínimo, um máximo, ou uma faixa numérica.
- As chaves simulam os seguintes metacaracteres: \* + ?.

## Metacaracteres tipo Âncora

Bem, deixando os quantificadores de lado, vamos agora falar sobre os metacaracteres do tipo âncora.

Por que âncora? Porque eles não casam caracteres ou definem quantidades, ao invés disso eles marcam uma posição específica na linha.

Assim, eles não podem ser quantificados, então o mais o asterisco e as chaves não têm influência sobre âncoras.

**Circunflexo:** O início

O nosso amigo circunflexo (êta nome comprido e chato) marca o começo de uma linha. Nada mais. `^[^oooo]`

**OBS:** Mas o circunflexo não é o marcador de lista negada?

Também, mas apenas dentro da lista (e no começo), fora dela ele é a âncora que marca o começo de uma linha, veja:

```
^[0-9]
```

Isso quer dizer; a partir do começo da linha, case um número, ou seja, procuramos linhas que começam com números. O contrário seria:

```
^[^0-9]
```

Ou seja, procuramos linhas que não começam com números. O primeiro circunflexo é a âncora e o segundo é o “negador” da lista. E como não podemos deixar de ser, é claro que o circunflexo como marcador de começo de linha só é especial se estiver no começo da ER. Não faz sentido procurarmos uma palavra seguida de um começo de linha, pois se tiver uma palavra antes do começo de uma linha, ali não é o começo da linha! Desse modo, a ER:

```
[0-9]^
```

Casa um número seguido de um circunflexo literal, em qualquer posição da linha. Com isso em mente, você pode me dizer o que casa a ER:

```
^^
```

Pois é, uma ER tão singela e harmônica como essa procura por linhas que começam com um circunflexo. Legal né? E para fechar, uma ER que em um e-mail, casa as conversas anteriores, aquelas linhas que começam com os sinais de maior >, abominados por muitos. Ei! Essa você mesmo pode fazer, não?

**Resumão:**

- Circunflexo é um nome chato, porém chapeuzinho é legal.
- Serve para procurar palavras no começo da linha.
- Só é especial no começo da ER (e de uma lista).

**Cifrão:** O fim

Similar e complementar ao circunflexo, o cifrão marca o fim de uma linha só é válido no final de uma ER. Como o exemplo anterior, `[0-9]$` casa linhas que terminam com um número. E o que você me diz da ER a seguir?

```
^$
```

Uma linha vazia.

Essa ER é sempre bom ter na manga, pois procurar por linhas em branco é uma tarefa comum nas mais diversas situações. Podemos também casar apenas os cinco últimos caracteres de uma linha.

```
.....$
```

Ou, ainda, que tal casarmos linhas que tenham entre 20 e 60 caracteres?

```
^.{20,60}$
```

É comum essas (inclusive eu) chamarem o cifrão de dólar. Vamos abolir essa prática. Chame ele de “ésse riscado”, mas dólar é feio. É como diria Júlio Neves, lugar de dólar é no bolso.

### Resumão

- Serve para procurar palavras no fim da linha.
- Só é especial no final da ER.
- É cifrão e não dólar.

### Borda: A limítrofe \b

A outra âncora que temos é a borda, que como o próprio nome já diz, marca uma borda, mais especificamente, uma borda de palavra.

Ela marca os limites de uma palavra, ou seja, onde ela começa e/ou termina. Muito útil para casar palavras exatas, e não partes de palavras. Veja como se comportam as ERs nas palavras dia, diafragma, melodia, radial e bom-dia:

#Expressão	Casa com
dia	dia,diafragma,melodia,radial,bom-dia!
\bdia	dia,diafragma,bom-dia!
dia\b	dia, melodia, bom-dia!
\bdia\b	dia,bom-dia!

Assim vimos que a borda forma um começo ou terminação de palavras.

Entenda que “palavra” aqui é um conceito que engloba [A-Za-z-0-9\_] apenas, ou seja, letras, números e o sublinhado. Por isso \bdia\b também casa bom-dia! pois o traço e a exclamação não são parte de uma palavra.

Ah! Dependendo do aplicativo, o sublinhamento não faz parte de uma palavra.

### Resumão

- A borda marca os limites de uma palavra.
- O conceito “palavra” engloba letras, números e o sublinhado
- A borda é útil para casar palavras exatas e são parciais.

**Escape:** A criptonita

E tudo estava indo bem na sua vida nova de criador de ERs, quando de repente...

Ó não preciso colocar um \* literal, o que fazer?

Se você está atento, lembrará que a lista tem suas próprias regras e que...

Dentro da lista, todo mundo é normal!

Precisou de um caractere que é um meta, mas você quer seu significado literal, coloque-o dentro de uma lista, então `lua[*] casa lua*`. O mesmo serve para qualquer outro metacaractere. Maaaaaaaas, para não precisar ficar toda hora criando listas de um único componente só para tirar seu valor especial, temos o metacaractere criptonita `\`, que “escapa” um metacaractere, tirando todos os seus poderes.

Escapando, `\*` é igual a `[*]` que é igual a um asterisco literal. Similarmente podemos escapar todos os metacaracteres já vistos.

```
\. \[ \] \? \+ \{ \} \^ \$
```

E para você ver como são as coisas, o escape é tão poderoso que pode escapar a si próprio! O caso uma barra invertida `\` literal. Ah! É claro, escapar um circunflexo ou cifrão somente é necessário caso eles estejam em suas posições especiais, como casar o texto `^destaque^`, em que ambos os circunflexos são literais, mas o primeiro será considerado uma âncora de começo de linha caso não esteja escapado.

Então agora que sabemos muito sobre ERs, que tal uma expressão para casar um número de RG no formato `n.nnn.nnn-n`?

```
echo "1.111.111-1" | egrep "^[0-9]{1}\.[0-9]{3}\.[0-9]{3}-[0-9]{1}$"
```

**Resumão**

- O escape escapa um metacaractere, tirando seu poder.
- `\*` = `[*]` = asterisco literal.
- O escape escapa o escape, escapando-se a si próprio simultaneamente.

**Ou:** O alternativo

É muito comum em uma posição específica de nossa ER termos mais de uma alternativa possível, por exemplo, ao casar um cumprimento amistoso, podemos ter uma terminação diferente para cada parte do dia:

```
echo "boa-tarde" | egrep "boa-tarde|boa-noite"
```

O `ou`, representado pela barra vertical `|`, serve para esses casos em que precisamos dessas alternativas. Essa ER se lê: “ou boa-tarde, ou boa-noite”, ou seja “ou isso ou aquilo”. Lembre que a lista também é uma espécie de `ou`, mais apenas para uma letra, então:

```
echo "gato" | egrep "[gpr]ato"
```

São similares, embora nesse caso em que apenas uma letra muda entre as alternativas, a lista é a melhor escolha. Em outro exemplo, o ou é útil também para casarmos um endereço de Internet, que pode ser uma página, ou um servidor FTP

```
echo "ftp://dominio.com.br" | egrep "http://|ftp://"
```

Ou isso ou aquilo, ou aquele outro... E assim vai. Pode-se ter tantas opções quantas se precise, Não deixe de conhecer o parente de 1º grau do ou, o grupo, que multiplica seu poder.

## Resumão

- O ou indica alternativas.
- Lista para um caractere, ou para vários.
- O grupo multiplica o poder do ou.

## Grupo: O pop

Assim como artistas famosos e personalidades que conseguem arrastar multidões, o grupo tem o dom de juntar vários tipos de sujeitos em um mesmo local. Dentro de um grupo podemos ter um ou mais caracteres, metacaracteres e inclusive outros grupos! Como em uma expressão matemática, os parênteses definem um grupo, e seu conteúdo pode ser visto como um bloco na expressão.

Todos os metacaracteres quantificadores que vimos anteriormente, podem ter seu poder ampliado pelo grupo, pois ele lhes dá mai abrangência. E o ou, pelo contrário, tem sua abrangência limitada pelo grupo, e pode parecer estranho, mas é essa limitação que lhe dá mais poder.

Em um exemplo simples, (ai)+ agrupa a palavra "ai" e esse grupo está quantificado pelo mais. Isso quer dizer que casamos várias repetições da palavra, como ai,ai,ai,ai,ai,... E assim podemos agrupar tudo o que quisermos, literais e metacaracteres, e quantificá-los:

Expressões	Casa com
(ha!)+	ha!,ha!ha!,ha!ha!ha!,...
(\.[0-9]){3}	.0.6.2,.2.8.9,.7.7.7,...
(www\.)?zz\.com	www.zz.com,zz.com

## Testando a expressão

```
echo ".zz.com" | egrep "(w?w?w?\.)?zz\.com"
```

E em especial nossa amigo ou ganha limites e seu poder cresce: Testes de expressão

```
echo "boa-tarde" | egrep "boa-(tarde|noite)"
```

## Teste de expressão

```
echo "incerto" | egrep "(in|con)?[sc]erto"
```

#Expressão	Casa com
boa-(tarde noite)	boa-tarde,boa-noite
(# n\. núm) 7	#7,n.7,núm 7
(in con)?certo	incerto,concerto,certo

Note que o grupo não altera o sentido da ER, apenas serve como marcador. Podemos criar subgrupos também, então imagine que você esteja procurando o nome de um supermercado em uma listagem e não sabe se este é um mercado supermercado ou um hipermercado.

```
(super|hiper)mercado
```

Consegue casar as duas últimas possibilidades, mas note que nas alternativas super e hiper temos um trecho per comum aos dois, então podíamos “alternativizar” apenas as diferenças su e hi:

```
(su|hi)permercado
```

Precisamos também casar apenas o mercado sem os aumentativos, então temos de agrupá-los e torná-los opcionais:

```
((su|hi)per)?mercado
#ou
(((su|hi)per)|mini)?mercado
```

Pronto! temos a ER que buscávamos e ainda esbanjamos habilidade utilizando um grupo dentro do outro.

E se eu tivesse minimercado também?

```
(mini|(su|hi|per))?mercado
```

E assim vai... Acho que já deu para notar quão poderosas e complexas podem ficar nossas ERs ao utilizarmos grupos, não? mas não acaba por aqui! Acompanhe o retrovisor na sequência.

Espera! E se eu quiser casar um par de parênteses literais?

Ah! Lembra-se do escape criptonita? Basta tirar o poder dos parênteses, escapando-os. Geralmente é preciso casar parênteses literais ao procurar por nomes de funções no código de um programa, como por exemplo Minha\_Funcao(). A ER que casa esta e outras funções é:

```
[A-Za-z0-9_]+\(\)
```

Ou ainda, caso acentuação seja permitida um nomes de função (lembra-se das classes POSIX!):

```
[[:alnum: _]]+\(\)
```

## Resumão

- Grupos servem para agrupar.
- Grupos são muito poderosos.
- Grupos podem conter grupos.
- Grupos são quantificáveis.

### Retrovisor: O saudosista

Já vimos o poder do grupo, e várias utilidades em seu uso. Mas ainda não acabou! Se prepare para conhecer o mundo novo que o retrovisor nos abre.

Ou seria mundo velho?

Ao usar um (grupo) qualquer, você ganha um brinde, e muitas vezes nem sabe. O brinde é o trecho de texto casado pela ER que está no grupo, que fica guardado em um cantinho especial, e pode ser usado em outras partes da mesma ER!

Como o nome diz, é retrovisor porque ele “olha para trás”, para buscar um trecho já casado. Isso é muito útil para casar trechos repetidos em uma mesma linha. Veja bem, é o trecho de texto, e não a ER.

Como exemplo, em um texto sobre passarinhos, procuramos o quero-quero. Podemos procurar literalmente por quero-quero, mas assim não tem graça, pois somos mestres em ERs e vamos usar o grupo e o retrovisor para fazer isso:

```
(quero)-\1
```

Então o retrovisor \1 é uma referência ao texto casado do primeiro grupo, nesse caso quero, ficando, no fim das contas, a expressão que queríamos. O retrovisor pode ser lembrado também como um link ou um ladrão, pois copia o texto do grupo.

Lembra que o escape \ servia para tirar os poderes do metacaractere seguinte.

Então, a essa definição agora incluímos: a não ser que este próximo caractere seja um número de 1 a 9, então estamos lidando com um retrovisor.

Notou o detalhe? Podemos ter no máximo 9 retrovisores por ER, então \10 é o retrovisor número 1 seguido de um zero. Alguns aplicativos novos permitem mais de nove.

O verdadeiro poder do retrovisor é quando não sabemos exatamente qual texto o grupo casará. Vamos estender nosso quero para “qualquer palavra”:

```
([A-Za-z]+)-\1
```

Percebeu o poder dessa ER? Ela casa palavras repetidas, separadas por um traço, como o próprio quero-quero, e mais: bate-bate, come-come etc. Mas e se tornássemos o traço opcional?

```
([A-Za-z]+)-?\1
```



Agora além das anteriores, nossa ER também casa bombom, lili, dudu, bibi e outros apelidos e nomes de cachorro.

Com uma modificação pequena, fazemos um minicorretor ortográfico para procurar por palavras repetidas como como estas em um texto:

```
([A-Za-z]+) \1
```

Mas como procuramos por palavras inteiras e não apenas trechos delas, então precisamos usar as bordas para completar nossa ER:

```
\b([A-Za-z]+) \1\b
```

Note como vamos contruindo as ERs aos poucos, melhorando, testando e não simplesmente escrevendo tudo de uma vez. Esta é a arte ninja de se escrever ERs.

### Mais detalhes

Como já dito, podemos usar no máximo nove retrovisores. Vamos ver uns exemplos com mais de um de nossos amigos novos:

#Expressão	Casa com
(lenta)(mente) é \2 \1	Lentamente é mente lenta
((band)eira)nte \1 \2a	bandeirante bandeira banda
in(d)ol(or) é sem \1\2	indolor é sem dor
((((a)b)c)d)-1= \1,\2,\3,\4	abcd-1= abcd,abc,ab,a

Para não se perder nas contagens, há uma dica valiosa: conte somente os parênteses que abrem, da esquerda para a direita. Este vai ser o número do retrovisor. E o conteúdo é o texto casado pela ER do parêntese que abre até seu correspondente que fecha.

**Atenção:** O retrovisor referencia o texto casado e não a ER do grupo. Nos nossos exemplos ocorre a mesma coisa porque a ER dentro do grupo já é o próprio texto, sem metacaracteres. Veja, entretanto, que `([0-9])\1` casa 66 mas não 69.

E se eu colocar um retrovisor em uma ER que não tem grupo?

Vai dar pau :)

Apenas como lembrete, algumas linguagens e programas, além da função de busca, têm a função de substituição. O retrovisor é muito útil nesse caso, para substituir “alguma coisa” por “apenas uma parte dessa coisa”, ou seja, extrair trechos de uma linha. Mais detalhes sobre isso adiante.

### Resumão

- O retrovisor só funciona se usados com o grupo.
- O retrovisor serve para procurar palavras repetidas.
- Numeram-se retrovisores contando os grupos da esquerda para direita.
- Temos no máximo 9 retrovisores por ER.

### Alguns exemplos de expressões

Expressão para validar endereço ipv4 então podemos ter ips como abaixo pois eles vão de 0.0.0.0 até 255.255.255.255:

```
0.0.0.0
10.10.10.10
199.199.199.199
220.220.220.220
255.255.255.255
```

Uma expressão para validar endereço ip abaixo.

```
^(1([0-9]?{2})|(2[0-5]?{2})|0{1})\.((1([0-9]?{2})|(2[0-5]?{2})|0{1})\.){2}(1([0-9]?{2}$)|(2[0-5]?{2}$)|0{1}$)
```

Como testar

```
echo "0.0.0.0" | egrep "^(1([0-9]?{2})|(2[0-5]?{2})|0{1})\.((1([0-9]?{2})|(2[0-5]?{2})|0{1})\.){2}(1([0-9]?{2}$)|(2[0-5]?{2}$)|0{1}$)"
0.0.0.0 #-> saída se houver casamento da er.
```

Mais um teste

```
echo "10.10.10.10" | egrep "^(1([0-9]?{2})|(2[0-5]?{2})|0{1})\.((1([0-9]?{2})|(2[0-5]?{2})|0{1})\.){2}(1([0-9]?{2}$)|(2[0-5]?{2}$)|0{1}$)"
10.10.10.10 #-> saída se houver casamento da er.
```

Mais um teste

```
echo "255.255.255.255" | egrep "^(1([0-9]?{2})|(2[0-5]?{2})|0{1})\.((1([0-9]?{2})|(2[0-5]?{2})|0{1})\.){2}(1([0-9]?{2}$)|(2[0-5]?{2}$)|0{1}$)"
255.255.255.255 #-> saída se houver casamento da er.
```

Agora vamos testar um endereço ip não valido 256.255.255.255 pois os ips ipv4 podem ir somente até 255.255.255.255

```
echo "256.255.255.255" | egrep "^(1([0-9]?{2})|(2[0-5]?{2})|0{1})\.((1([0-9]?{2})|(2[0-5]?{2})|0{1})\.){2}(1([0-9]?{2}$)|(2[0-5]?{2}$)|0{1}$)"
#Pode ser notado que não temos saída alguma pois não houve casamento da er.
```

Agora vamos criar uma er para fazer validação de endereço de email

Vamos supor que o endereço de email pode começar com uma letra ou um \_ e tem que ter no mínimo um caractere antes do @ e após o @ do a domínio vai ter que ter no mínimo 4 caracteres e no máximo 15 e pode ser .com ou .com.br ou ainda somente .br vamos a expressão.

```
^([[:alnum:]]|_|)+([[:alnum:]]|_|.){0,255}@([[:alnum:]]{4,15})\.+([[:alnum:]]{2,3})\.?([[:alpha:]]{2}?)$
```

Vamos efetuar alguns testes

```
echo "aR2_email@gmail.com.br" | egrep '^([[:alnum:]]|_|)+([[:alnum:]]|_|.){0,255}@([[:alnum:]]{4,15})\.+([[:alnum:]]{2,3})\.?([[:alpha:]]{2}?)$'
```

```
aR2_email@gmail.com.br #-> saída do casamento da er.
```

Mais um testes agora de um email que comece com `_` e que o domínio termine com `.br`

```
echo "_email@gmail.br" | egrep '^([[:alnum:]]|_)+([[:alnum:]]|_|.){0,255}@([[:alnum:]]{4,15})\.+([[:alnum:]]{2,3})\.?([[:alpha:]]{2}?)$'
_email@gmail.br #-> saída do casamento da er
```

Último teste agora o email vai começar com um número e vai terminar com `.com`

```
echo "1email@gmail.com" | egrep '^([[:alnum:]]|_)+([[:alnum:]]|_|.){0,255}@([[:alnum:]]{4,15})\.+([[:alnum:]]{2,3})\.?([[:alpha:]]{2}?)$'
1email@gmail.com #-> saída do casamento da er
```

Agora vamos a um último exemplo vamos efetuar validação de um CPF que segue a lógica de 111.111.111-11

```
^([[:digit:]]{3})\.([[:digit:]]{3})\.([[:digit:]]{3})-([[:digit:]]{2})$
```

Vamos testar a nossa expressão

```
echo "111.222.333-44" | egrep "^([[:digit:]]{3})\.([[:digit:]]{3})\.([[:digit:]]{3})-([[:digit:]]{2})$"
111.222.333-44 #-> saída do casamento da er
```

## Referências

1. <http://www.piazinho.com.br/> [http://www.piazinho.com.br/]
2. [http://pt.wikipedia.org/wiki/Express%C3%A3o\\_regular](http://pt.wikipedia.org/wiki/Express%C3%A3o_regular) [http://pt.wikipedia.org/wiki/Express%C3%A3o\_regular]

Exceto onde for informado ao contrário, o conteúdo neste wiki está sob a seguinte licença: CC Attribution-Share Alike 3.0 Unported [http://creativecommons.org/licenses/by-sa/3.0/]

Contact : [webmaster@neolao.com](mailto:webmaster@neolao.com)